**A group of white letters and a circle with a black background

Description automatically generated**

**Game Software Design Document**

AAA(A) Group

Zack Cornfield, Jacob Mondi, Milad Rakhshbahar, Thomas Stiles

GitHub Repo: <https://github.com/stiletj/AAAAEngine>

Contents

[**Introduction** 3](#_Toc163928989)

[**Game Engine Overview** 3](#_Toc163928990)

[**System Requirements** 3](#_Toc163928991)

[**Game Engine User Guide** 4](#_Toc163928992)

[**Game Engine APIs/Librarys** 5](#_Toc163928993)

[**ENTT (Entities and Components)** 6](#_Toc163928994)

[**Directions on Adding Entities** 6](#_Toc163928995)

[**Directions on Adding Components** 6](#_Toc163928996)

[**Asset Information** 7](#_Toc163928997)

[**How to load Assets** 7](#_Toc163928998)

[**Physics Simulation** 8](#_Toc163928999)

[**Adding Physics Components to Entities** 8](#_Toc163929000)

[**Input Handling** 9](#_Toc163929001)

[**Directions on Using the GLFW Input System:** 9](#_Toc163929002)

[**Scripting** 10](#_Toc163929003)

[**Debugging** 11](#_Toc163929004)

[**Further Documentation (External Documentation)** 12](#_Toc163929005)

[**Licensing** 13](#_Toc163929006)

# Introduction

This is the Game Software Design Document for AAAAEngine. This document provides resources for understanding and utilizing our game engine software effectively.

In the sections ahead, you’ll find detailed explanations covering our engine’s architecture, system requirements, user guide, scripting capabilities, debugging mechanisms, and more.

# Game Engine Overview

The game engine has been created using an Entity Component System, which allows for the creation and management of entities and their respective components. The engine is fulling interacted with by the user through scripts. A Game script is created, which can then be used to add entities. Each entity has components. Some components have been created for the user (Transform, RigidBody or Colliders for example). Other components can be created by the user through scripts.

The engine is fully code/text based.

# System Requirements

* Windows 10/11
* Support for LUA and C++

# Game Engine User Guide

There are a limited number of predefined types natively available in Lua such as:

* Integer
* Number (double-precision floating point)
* String
* Boolean
* Function

For ease of understanding and efficient use of the engine we have provided multiple custom types for Lua, such as:

* ENTITY (entt::entity facade)
* BodyType
* Vector3
* GUKey
* GUMouse

These can be used in conjunction with the numerous custom functions and classes that we have provided Lua, to develop comprehensive games.

The Engine also provides users with many in-built systems such as:

* Physics System
* Input Handling System
* Terrain System
* Rendering System
* Skybox System
* Texturing and Material System
* Entity Management System

These can be utilised to develop a game easier within one or more Lua files.

# Game Engine APIs/Librarys

* React Physics 3D
* ImGUI
* OpenGL(3) and GFLW
* ENTT
* Assimp
* GLM
* GLEW
* Lua
* Sol

# ENTT (Entities and Components)

## **Directions on Adding Entities**

* If the user wishes to use this entity outside of the Start function, then they will need to create a variable of type ENTITY to store it:
  + entity = ENTITY:new()
* Then, to add an entity inside a Lua script the user must first either navigate to or create the Start function. This will ensure the command will be processed in the correct order in relation to the rest of the engine.
* Now the user can create the entity and store it in their new ‘entity’ variable:
  + entity = CreateEntity(Transform transform, bool torender)
* The user must pass a variable of type Transform and one of type bool to the function. A Transform is a structure that stores 3 Vector3s:
  + Position
  + Rotation
  + Scale

transform = Transform:new()

transform:Set(1, Vector3) --Position

transform:Set(2, Vector3) --Rotation

transform:Set(3, Vector3) --Scale

* And the bool to specify if the entity is to be active or not.

## **Directions on Adding Components**

* If the user wishes to add a component to one of their entities, they require an existing entity first.
* Then, to add the component the user must specify the type of component, by choosing the correct function, as well as the constructor argument values that are required.
* For example, if the user wanted to add a PhysicsBody to an Entity:
  + ***AddPhysicsBody(entity)***
* Or add a Camera to an Entity:
  + ***AddCamera(entity, position, lookAt, upVector, pitch, yaw)***

# Asset Information

* Assets should only be of OBJ File format/s
* Textures should only be of JPG/PNG File format/s

## **How to load Assets**

* To load an asset the user will use the MeshLoader which provides 2 functions:
  + ***LoadMesh(“filename”)***
  + ***GetMesh()***
* The user can use LoadMesh in the Start function to load the mesh into the MeshLoader.
* Then the user can add a Mesh component to an existing entity with the Mesh stored in the MeshLoader by using GetMesh() as an argument:
  + ***AddMesh(entity, GetMesh())***
* This can be repeated over and over with the same MeshLoader as the new information overrides the old information.

# Physics Simulation

## **Adding Physics Components to Entities**

* Physics simulation is performed through ReactPhysics3D, which provides a Physics World, Rigid Bodies, and Collision Shapes.
* A PhysicsBody component can be added to an existing Entity which stores the Entity’s Physics object:
  + ***AddPhysicsBody(entity)***
* To add a Rigid body to this Entity the user can use the function ‘setEntityPhysicsBody’ which takes an existing Physics object as input:
  + ***setEntityPhysicsBody(entity, Physics)***
* The Physics object can be created with a Rigid body with the ‘AddRigidbody’ function:
  + ***Body = Physics:new()***
  + ***Body:AddRigidbody(position, mass, body type, gravity)***

# Input Handling

* Input Handling is done through GLFW and its façade.
* There exists an InputHandler that uses GLFW to take an input.
* But it reduces the calculations done in Lua by providing functions that take Lua functions as input and call them when the event triggers.

## **Directions on Using the GLFW Input System:**

* The user must first create a void function with no input arguments that are to be used when the specific event is triggered:

Function PressW()

--Trigger resolution code here

End

Function MoveMouse()

--Trigger resolution code here

End

* Then if the function to be used on the press of a key (PressW) the user must bind a key using the ‘BindKey’ function: ***BindKey(GUKey.Key\_W, PressW)***
* The first argument is of type GUKey and specifies the key that is to be pressed. The second argument is the function to be used once the event is triggered.
* Then, if the function is to be used on the press of a mouse button, the ‘BindMouse’ function is used: ***BindMouse(GUMouse.MOUSE\_BUTTON\_1)***
  + The first argument is of type GUMouse, and the second is the function called when the event is triggered.
* Then, if the function is to be used when the mouse is moved, the user can use the function ‘MousePosFunc’. This just takes the function: ***MousePosFunc(MouseMove)***

# Scripting

* Scripting is done using Lua.
* Currently only the base library is active in Lua.
* Numerous classes and functions have been bound to all Lua scripts that are added to the project by the user.
* The user is provided with an initial main script which is used for all of the overall control of the project.
* The main script contains the Start function and the Update function:
* The Start function is called just before the beginning of the game loop.
* The Update function is called in the game loop after all the updates have been done for physics and time.
* Although there is one main script, the user can create new scripts by saving a text file as .lua, they can then add it to their project with the ‘AddScript’ function:

AddScript(“filename.lua”)

* However, all scripts must be stored in the ‘scripts’ directory inside the working directory, to force good file management.
* The user can create subdirectories inside the ‘scripts’ directory although, the user must specify these file paths when adding the script:

AddScript(“filepath/filename.lua”)

* In Lua, variables that are defined in a function are local to that function.
* If they are not defined in a function, then they are global variables. But there are two different types of global variables:
  + Local – Can only be used in the script they are defined in.
  + Global – Can be used in any script.
* The functions, classes and custom types that have been bound to Lua can be used in any script that the user adds via the main Lua script.

# Debugging

Currently, there is a limited debugging system for Lua, the user will get syntax errors when their syntax is wrong. However, they will not always get error messages if their logic is incorrect.

In terms of the Engine, most areas have reasonably comprehensive debugging. Error handling is done in most cases through the use of print statements, emergency exiting of the program/functions, assert statements, or try-catch statements.

# Licensing

* Geomipmapping CLOD terrain code references: <https://www.youtube.com/watch?v=08dApu_vS4c&ab_channel=OGLDEV>
* Lua binding:

<https://www.youtube.com/watch?v=n-7NrWv-PAw>

<https://github.com/tiger-punch-sports-club/lua-sol2-tutorial>